University of Saskatchewan
Department of Computer Science

Cmpt 330
Final Examination

December 21, 2001

Time: 3 hours
Total Marks: 90

Professor: A. J. Kusalik
Closed Book†

Name: _____

Student Number: _____

Directions:

Answer each of the following questions in the space provided in this exam booklet. If you must continue an answer (e.g. in the extra space on the last page, or on the back side of a page), make sure you clearly indicate that you have done so and where to find the continuation.

Ensure that all answers are written legibly; no marks will be given for answers which cannot be deciphered. Where a discourse or discussion is called for, please be concise and precise. Do not give "extra answers". Extra answers which are incorrect may result in your being docked marks.

Use of calculators during the exam is not allowed.

If any question requires an assumption as to a particular operating system context, assume UNIX as manifest by NetBSD. If you find it necessary to make any other assumptions to answer a question, state the assumption with your answer.

Marks for each major question are given at the beginning of that question.

Good luck.

For marking use only:

A.	_____/5	E.	_____/6	I.	_____/7
B.	_____/10	F.	_____/11	J.	_____/4
C.	_____/11	G.	_____/6	K.	_____/11
D.	_____/14	H.	_____/5		

Total: _____/90

---

† Closed book, except for one optional 8.5×11 inch quick reference sheet ("cheat sheet") of the student's own compilation.

A. (5 marks)

The UNIX "*man* pages" are valuable for getting information about using, and programming for, the UNIX operating system. The *man* pages are organized into sections, where particular types of information (i.e. *man* pages on a specific general topic) are in a given section. For example, section 4 of the *man* pages is devoted to special files and hardware support. Thus, the *man* pages describing the *wd* disk driver and `tty` devices are in section 4. As another example, man pages for games are in section 6.

For each of the topic areas below, say what section of the *man* pages covers that topic area.

____ file formats

____ general user commands (tools and utilities)

____ system maintenance and operation commands

____ system (kernel) calls

____ C library functions

B. (10 marks)

Consider the following common directories in BSD UNIX. Match the pathnames with their appropriate descriptions; match each directory name with the (single) description which is most appropriate. Indicate the matches by a line drawn between the directory name and the appropriate description. Each name is to match one and only one description.

directory or file		**is or contains**
/lost+found	■	■ system "scratch files" directory
/dev	■	■ an infinite source of "end-of-file" indications
/tmp	■	■ basic executable commands such as *rm*, *cp*, *sh*, *csh*, *ed*, *echo*, *test*
/bin	■	■ special files (device interfaces)
/etc	■	■ directory in every file system root to hold detached files found by *fsck*
/dev/null	■	■ standard "include" header files
/usr/include	■	■ majority of UNIX user commands
/usr/bin	■	■ file descriptor files
/kern	■	■ system administration data and commands
/dev/fd	■	■ interface to various fields in the kernel's memory space

C. (1 mark each= 11 marks)

Answer each of the following questions with a very short, precise answer.

1. A user needs to write a UNIX shell script which contains a "here document". What redirection symbol is used to implement a "here document"?

2. Consider the following transcript of a user's interaction with UNIX:

```
tonka4(15)> ical & exmh &
[1] 14596
[2] 14597
```

What are the two numbers `14596` and `14597`?

3. Give the symbol name for a signal which, when sent to a process, will cause abnormal termination of that process.

4. Give the name of one `struct` or macro defined in `<ufs/ffs/fs.h>` on one of the *tonka* machines.

5. Three components of disk access time are seek time, latency, and head-switch time. Interleaving policies (for layout of blocks of a file on the disk) were introduced into the BSD FFS (Fast File System) to reduce which component of disk access time?

6. Which is a more likely value for the free-space reserve for a FFS on a BSD UNIX system, 5% or 95%?

7. What is the "magic number" that indicates that a file contains (is) a script? Give that "magic number" below.

8. What is the kernel call used to deal with device-dependent characteristics of peripheral devices, or perform device-dependent operations on them?

9. Suppose a UNIX system has a physical disk which has been partitioned. Further assume that the disk is not being used for any other operating system (e.g. there is no partitioning for BIOS). If a bootstrap is to be written to the disk, which partition is best to use?

10. One type of terminal line discipline is "cbreak". Give the name of another terminal line discipline.

11. Name a computer network that connected major American universities and research centers prior to the advent of the Internet.

D. (2 marks each = 14 marks)

Answer each of the following questions with a short, precise answer.

1. What are (were) the two main "streams" in the development of UNIX? I.e. what are the two main "flavors" of UNIX which have resulted from the manner in which the operating system evolved?

2. The include file <ufs/ufs/dir.h> defines the following two macros:

```
#define IFTODT(mode)    (((mode) & 0170000) >> 12)
#define DTTOIF(dirtype) ((dirtype) << 12)
```

The program *fsdb(8)* can be used to display the content of inodes in a human readable form. Suppose that program generates the following when executed on one of the *tonka* machines in the NetBSD lab:

```
fsdb (inum: 2)> inode 11
I=11 MODE=120755 SIZE=11
        MTIME=Sep  2 15:41:34 2000 [0 nsec]
        CTIME=Aug 23 07:50:33 2000 [630000000 nsec]
        ATIME=Sep  2 15:41:34 2000 [0 nsec]
OWNER=root GRP=wheel LINKCNT=1 FLAGS=0x0 BLKCNT=0x0 GEN=0x1
```

The directory entry for this inode will also record the type of the inode. One can convert between the type as stored in the inode and the type stored in the directory entry using the two macros from <ufs/ufs/dir.h> given earlier.

Using the appropriate macro defined above, give the value of the type field (for this inode) as stored in a directory entry for this inode. Make sure you indicate if your value is in decimal, octal, hexadecimal, or binary.

3. A UNIX user wishes to give timing results for her program. Give the names of two kernel calls or C library functions related to time and timing which she might use.

4. Suppose you obtain the following output from `ls` when executed on a BSD UNIX system:

```
tonka5> ls -l /tmp/strange
srw-rw-rw-  1 root  wheel            0 Nov 23 03:04 a
lrwxrwxrwx  1 root  wheel            3 Nov 23 05:40 b -> ../foobar
-rw-r--r--  1 root  wheel            4 Nov 23 03:04 c
srwxrwxrwx  1 root  wheel            0 Nov 23 03:04 d
brw-r-----  1 root  operator  6,     0 Jul 30 14:58 e
brw-r-----  1 root  operator  6,     1 Jul 30 14:58 f
crw-------  1 root     wheel  12,    0 Jul 30 14:58 g
drwxr-xr-x  2 root  wheel         2048 Jul 13 23:06 h
```

i) Which of the above files are (is a) character special file(s)? Give the file name(s).

ii) What is the major unit number of the block special file or files above? Give that number.

5. Name 2 connectionless TCP/IP protocols above the data link layer.

6. Suppose that instead of using 16 bits for the network ID in a class B address, 20 had been used. How many class B networks would have been possible then, accounting for IDs which are "reserved"? You can give your answer as an expression. Assume that the same leading bits would indicate a class B address.

7. Name 2 types of sockets possible in the AF_INET domain.

E. (6 marks)

Suppose that the output of the *mount(8)* command on a UNIX system is as follows:

```
csh> mount
/dev/wd0a on / type ufs (local)
/dev/wd0e on /usr type ufs (local)
barbie.usask.ca:/home on /usr/home type nfs
cs:/var/spool/mail on /var/spool/mail type nfs
skorpio3:/faculty/kusalik on /usr/home/kusalik type nfs
```

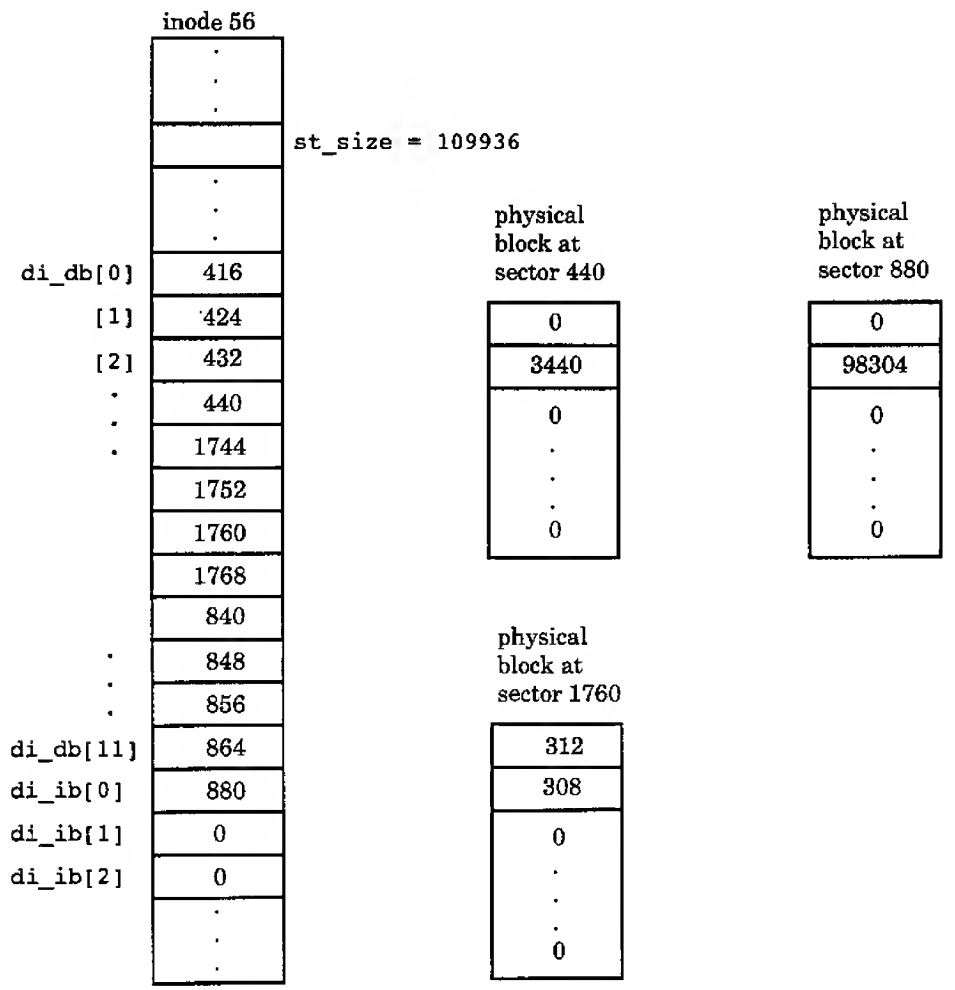
I.e. the file systems mentioned in the output above are currently mounted, and part of the overall file system.

Draw a diagram — it should have the form of a tree — which accurately represents the overall file system structure on this UNIX system. Be sure to indicate individual component file systems (which have been mounted) and mount points. Also identify in your diagram where each constituent file system originates.

F. (2+2+3+4 = 11 marks)

Consider a BSD UNIX file system under the following assumptions and conditions:

- Data block / fragment block sizes are 8192/1024.
- The disk storing the file system has 512-byte sectors.
- File system block addresses (values of type `ufs_daddr_t`) are 4 bytes in length.
- `NDADDR` = 12 (i.e. there are 12 direct block addresses in an inode) and `NIADDR` = 3 (i.e. there are at most 3 indirect block addresses in an inode).
- Some of the information on the disk is described by the following diagram (specifically, the diagram describes storage related to inode 56):



- In the diagram, `di_db` is the vector of `NDADDR` direct block pointers, and `di_ib` is the vector of `NIADDR` indirect block pointers. Fields are shown interpreted as 32-bit unsigned decimal

integers.

- As shown in the diagram, the size of the file described by inode 56 is 109936 bytes.

In the BSD FFS, file system block addresses are the same as fragment blocks (e.g. a direct block pointer value of 22 in an inode indicates fragment block 22). However, fragment block addresses (numbers) are different than physical block addresses (i.e. sector numbers).

Based on the above assumptions and descriptions, answer the following questions. Note that subsequent to the questions is supplementary information that may be useful in performing calculations.

1. What physical block stores the byte at each of the following byte addresses within the file described by inode 56? I.e. give the number of the sector that is used to store the byte of the file at each specified address.

   a) address 880

   b) address 25000

2. Does the file described by inode 56 contain a hole? If so, give a byte address (within the file) that falls within the hole.

3. Does the file contain a fragment? If so, how large is the fragment, both in terms of bytes and fragment blocks?

4. Examination of the free-space map for the file system shows that fragment block 304 is allocated to a file. Is the information stored in file system block (fragment block) 304 on the disk necessarily part of a (whole) datablock or a fragment? How do you know?

Since calculators are not allowed for this exam, the following information is being made available to assist in performing arithmetic calculations.

```
512 x 2 = 1024          1024 x 2 = 2048          8192 x  2 =  16384
512 x 3 = 1536          1024 x 3 = 3072          8192 x  3 =  24576
512 x 4 = 2048          1024 x 4 = 4096          8192 x  4 =  32768
512 x 5 = 2560                                   8192 x 11 =  90112
512 x 6 = 3584           304 x 2 = 608           8192 x 12 =  98304
512 x 7 = 3584                                   8192 x 13 = 106496
                                                 8192 x 14 = 114688

312/8 = 39              512/4 = 128              8192/512 = 16
304/8 = 38                                       8192/8 = 1024
                                                 8192/4 = 2048

880 - 512 = 368                          11631 - 8192 = 3439
                                         11632 - 8192 = 3440
12600 - 8192 = 4408
                                         109935 - 98304 = 11631
                                         109936 - 98304 = 11632

24576 + 512 = 25088
```

The following extra space is also being made available for your rough calculations.

G. (3+3=6 marks)

Each of the following questions involves a systems programming or operations scenario. Answer each with a precise and concise answer based on your systems programming knowledge and experience. Note that *man* page information for relevant system or library functions complete each problem statement.

1. A programmer is trying to write a program in which he or she needs to do a *stat(2)* call to obtain inode information for a file. The program compiles without errors or warnings. However, when the programmer tries to execute the program it "crashes". More specifically, the outline of the program is follows:

```
main(int argc, char **argv)
.....
struct stat *st=NULL;
.....
if(stat(argv[1],st)<0)
   perror(argv[1]);
......
```

and when invoked by

```
./a.out myfile
```

it yields

```
myfile: Bad address
Segmentation fault (core dumped)
```

What is the error? How would you fix it? Indicate your fix by a minimal modification to the code sample above.

If you do not recall how *stat(2)* works, select portions of the man page for the kernel call are given below.

```
NAME
     stat, lstat, fstat - get file status

SYNOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>

     int
     stat(const char *path, struct stat *sb);

DESCRIPTION
     The stat() function obtains information about the file pointed to by
     path.  Read, write or execute permission of the named file is not re-
     quired, but all directories listed in the path name leading to the file
```

```
must be searchable.

The sb argument is a pointer to a stat() structure as defined by
<sys/stat.h> (shown below) and into which information is placed concern-
ing the file.

RETURN VALUES
     Upon successful completion a value of 0 is returned.  Otherwise, a value
     of -1 is returned and errno is set to indicate the error.

ERRORS
     stat() and lstat() will fail if:

     [ENOTDIR]      A component of the path prefix is not a directory.

     [ENAMETOOLONG]
                    A component of a pathname exceeded {NAME_MAX} characters,
                    or an entire path name exceeded {PATH_MAX} characters.

     [ENOENT]       The named file does not exist.

     [EACCES]       Search permission is denied for a component of the path
                    prefix.

     [ELOOP]        Too many symbolic links were encountered in translating the
                    pathname.

     [EFAULT]       sb or name points to an invalid address.

     [EIO]          An I/O error occurred while reading from or writing to the
                    file system.
```

2. A student is writing a client/server program pair. Unfortunately, for some reason the server seems to "hang" after it accepts a connection on its socket. The relevant portion of code of the server program is as follows:

```
len = sizeof(struct sockaddr_in);

/*
 * Bind the socket to the address.
 */
if (bind(listensockfd, (struct sockaddr *) &server, len) < 0) {
    perror("bind");
    exit(1);
}

/*
 * Listen for connections.
 */
if (listen(listensockfd, 5) < 0) {
    perror("listen");
    exit(1);
}

/*
 * Accept a connection.
 */
```

```
if (newsockfd = accept(listensockfd, (struct sockaddr *) &server,
                       &len) < 0) {
    perror( "accept" );
    exit( 4 );
}

/*
 * Read first part of protocol from the client
 */
n = recv(newsockfd, buf, sizeof(buf), 0);
```

The variables `len`, `newsockfd`, and `listensockfd` are declared to be of type `int`. Memory space is allocated earlier in the program for the `sockaddr_in struct` named by `server` and for the buffer pointed to by `buf`. The variable `buf` is of type pointer-to-char.

Using *ddd(1)* and other tools, the student has characterized the bug as follows. After evaluation of the `if`-statement containing the call to *accept(2)*, the value in `newsockfd` is found to be 0 (the file descriptor for standard input) instead of a previously unused file descriptor. File descriptor 0 is already in use prior to the *accept()* call, being used for the standard input. The variable `listensockfd` is set to a reasonable value prior to the *accept()* call and is unchanged by *accept()*. The expression involving *accept()* within the `if`-statement evaluates to false (value of 0) and the error-reporting branch is not taken. Finally, the call to *recv()*, to get a first communication using `newsockfd`, never returns even though data is successfully sent to the server by a client.

What error has been made in the above program? How would you fix it? Indicate your fix by a minimal modification to the code sample above.

Hint: the problem is not with the call to *recv()*.

If you do not recall how *accept(2)* works, select portions of the man page for the kernel call are given below.

```
NAME
     accept - accept a connection on a socket

SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>

     int
     accept(int s, struct sockaddr *addr, socklen_t *addrlen);

DESCRIPTION
     The argument s is a socket that has been created with socket(2), bound to
     an address with bind(2), and is listening for connections after a
     listen(2).  The accept() argument extracts the first connection request
     on the queue of pending connections, creates a new socket with the same
```

properties of s and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, accept() blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept() returns an error as described below. The accepted socket may not be used to accept more connections. The original socket s remains open.

The argument addr is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the addr parameter is determined by the domain in which the communication is occurring. The addrlen is a value-result parameter; it should initially contain the amount of space pointed to by addr; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

RETURN VALUES

The call returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

H. (5 marks)

When displaying a file to the screen, using *more(1)* for example, it is wise to first make sure that the file contains text rather than binary data. Suppose that a user wishes to invoke the *more(1)* command on the file important_file in her current working directory. Give a single complex UNIX command which will invoke *more(1)* on the file important_file only if the file contains text; i.e. only if the output from *file(1)*, when invoked with important_file as an argument, produces a string which contains the substring "text". Give the command in either *sh* (Bourne shell) or *csh*.

Note: do not give a script. Rather, give a single complex command.

Hint: the command

```
test -n s1
```

exits with success if the length of *s1* is not zero.

I. (7 marks)

A programmer needs a C function which will accept the name of a file as an argument, open that file for append access, and redirect file descriptor 0 (standard input) to that file. In the space below, write such a C function. The function has only one argument, and it is of type pointer-to-char. The argument points to a character array (a string) in which the name of the file to be opened has been stored. The string is null-terminated. The function's return value is of type int. The return value is 1 if the

function is successful. If any of the system or library calls within the function encounters an error, an appropriate error is to be generated on the stream `stderr` and the function immediately returns with a value of zero. At the beginning of your function, specify all the "include files" that are necessitated by library and system calls within your function. Also make sure to close within your function any file descriptors that are no longer needed.

Portions of *man* pages from various system and library calls which you may find useful are given below.

SYNOPSIS

```
#include <unistd.h>

int
close(int d);
```

DESCRIPTION

The close() system call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

SYNOPSIS

```
#include <unistd.h>

int
dup(int oldd);

int
dup2(int oldd, int newd);
```

DESCRIPTION

dup() duplicates an existing object descriptor and returns its value to the calling process (newd = dup(oldd)). The argument oldd is a small non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by getdtablesize(3). The new descriptor returned by the call is the lowest numbered descriptor currently not in use by the process.

In dup2(), the value of the new descriptor newd is specified. If this descriptor is already in use, the descriptor is first deallocated as if a close(2) call had been done first.

RETURN VALUES

The value -1 is returned if an error occurs in either call. The external variable errno indicates the cause of the error.

SYNOPSIS

```
#include <stdio.h>

int
fclose(FILE *stream);
```

DESCRIPTION

The fclose() function dissociates the named stream from its underlying file or set of functions. If the stream was being used for output, any buffered data is written first, using fflush(3).

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, EOF is returned and

the global variable errno is set to indicate the error. In either case no further access to the stream is possible.

SYNOPSIS

```
#include <stdio.h>

FILE *
fopen(const char *path, const char *mode);

FILE *
fdopen(int fildes, const char *mode);

FILE *
freopen(const char *path, const char *mode, FILE *stream);
```

DESCRIPTION

The fopen() function opens the file whose name is the string pointed to by path and associates a stream with it.

The argument mode points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

``r'' Open text file for reading.

``r+'' Open for reading and writing.

``w'' Truncate file to zero length or create text file for writing.

``w+'' Open for reading and writing. The file is created if it does not exist, otherwise it is truncated.

``a'' Append; open for writing. The file is created if it does not exist.

``a+'' Append; open for reading and writing. The file is created if it does not exist.

The fdopen() function associates a stream with the existing file descriptor, fildes. The mode of the stream must be compatible with the mode of the file descriptor. The stream is positioned at the file offset of the file descriptor.

The freopen() function opens the file whose name is the string pointed to by path and associates the stream pointed to by stream with it. The original stream (if it exists) is closed. The mode argument is used just as in the fopen() function. The primary use of the freopen() function is to change the file associated with a standard text stream (stderr, stdin, or stdout).

RETURN VALUES

Upon successful completion fopen(), fdopen() and freopen() return a FILE pointer. Otherwise, NULL is returned and the global variable errno is set to indicate the error.

```
SYNOPSIS
     #include <fcntl.h>

     int
     open(const char *path, int flags, mode_t mode);

DESCRIPTION
     The file name specified by path is opened for reading and/or writing as
     specified by the argument flags and the file descriptor returned to the
     calling process.  The flags are specified by or'ing the following values.
     Applications must specify exactly one of the first three values (file ac-
     cess modes):

          O_RDONLY    Open for reading only.

          O_WRONLY    Open for writing only.

          O_RDWR      Open for reading and writing.

     Any combination of the following may be used:

          O_NONBLOCK  Do not block on open or for data to become available.

          O_APPEND    Append to the file on each write.

          O_CREAT     Create the file if it does not exist, in which case the
                      file is created with mode mode as described in chmod(2)
                      and modified by the process' umask value (see
                      umask(2)).

          O_TRUNC     Truncate size to 0.

          O_EXCL      Error if O_CREAT and the file already exists.

     If successful, open() returns a non-negative integer, termed a file de-
     scriptor.  It returns -1 on failure.  The file pointer used to mark the
     current position within the file is set to the beginning of the file.
```

```
SYNOPSIS
     #include <stdio.h>

     void
     perror(const char *string);

DESCRIPTION
     The perror() function looks up the language-dependent error message
     string affiliated with an error number and writes it, followed by a new-
     line, to the standard error stream.

     If the argument string is non-NULL it is prepended to the message string
     and separated from it by a colon and a space (`: ').  If string is NULL
     only the error message string is printed.

     The contents of the error message string is the same as those returned by
     strerror() with argument errno.
```

J. (4 marks)

In the space below, write a shell script for NetBSd that will produce on the standard output, a count of the number of processes that currently exist on the system. The script, however, cannot use the *ps(1)*, *systat(1)*, or *top(1)* commands. You can assume that `/etc/fstab` on the system you are using contains the line

```
/proc   /proc   procfs   rw     0   0
```

The script does not take a command-line argument.

K. (3+4+4+ = 11 marks)

Answer each of the following questions with a discussion-oriented answer.

1. Give the name of a long-lived process typical of BSD UNIX systems. What function or service is provided by the program or process you have specified?

1. Suppose a process (the "parent process") creates a child process via a *fork()*. Further assume that the child process subsequently terminates prior to the parent process terminating. Two ways in which the child process can terminate are by executing an *exit()* or by receiving a signal which causes termination. The parent process can determine how a child terminated. That is, the parent

process can determine if the child terminated due to executing `exit()` or because to received a signal. How can the parent process do this? What kernel calls does it need to execute? What parameters in/from those kernel calls does it need to examine? What fields within those parameters are relevant, and what values are indicative?

3. Which uses up more space on a BSD UNIX FFS: a symbolic link or a hard link? Why?

Epilog

That's it! You're all done!

I sincerely hoped that you learned a lot in this course, and will find the material covered in the course valuable in your future endeavors. Have a merry Christmas!